

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедры МО ЭВМ

ОТЧЕТ
по лабораторной работе №1
по дисциплине «Алгоритмы и структуры
данных»

Тема: Рекурсия

Студентка гр. 8304

Мельникова О.А.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2019

Цель работы

Ознакомиться с основными понятиями и приемами рекурсивного программирования, получить навыки программирования рекурсивных процедур и функций на языке программирования C++.

Задание

Вариант 2

Задано конечное множество имен жителей некоторого города, причем для каждого из жителей перечислены имена его детей. Жители X и Y называются родственниками, если (а) X – ребенок Y , (б) либо Y – ребенок X , либо существует некоторый Z , такой, что X является родственником Z , а Z является родственником Y . Перечислить все пары жителей города, которые являются родственниками.

Считывание и функция GetManID

Считывание выполняет функция `ReadAndWritePeople`, которая принимает на вход вектор имен и два вектора индексов, показывающих родство, а также количество элементов в этих векторах.

Из файла в цикле считываются строки с именами, где первое имя – это родитель, а последующие – дети. При считывании имен происходит поиск в векторе с именами, и если текущего имени там нет, то записываем его. Кроме того заполняем векторы индексов (первое имя – каждое последующее). Для быстрого поиска индекса в векторе имен создана функция `GetManID`.

Затем выводится список всех жителей и таблица отношений по индексам вектора имен, показывающая родство.

Функции GetRoot и UnionBranch

Функции созданы для поиска системы непересекающихся множеств.

Пускай мы оперируем элементами N видов (для простоты, здесь и далее — числами от 0 до $N-1$). Некоторые группы чисел объединены в множества.

Также мы можем добавить новый элемент, он тем самым образует множество размера 1 из самого себя. И наконец, периодически некоторые два множества нам потребуется сливать в одно.

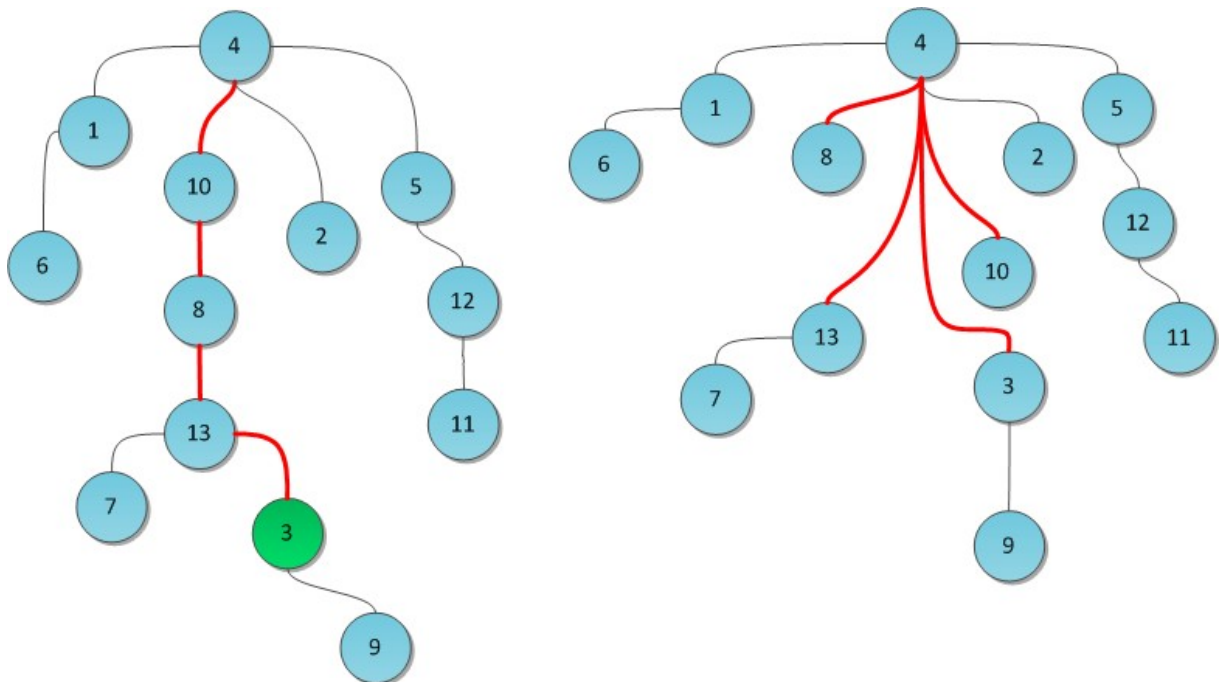
Чтобы создать новое дерево из элемента X , достаточно указать, что он является корнем собственного дерева, и предка не имеет $p[x] = x$, это будет сделано в `main()`.

GetRoot()

Создан для возвращения *идентификатора* множества, которому принадлежит элемент X . В качестве идентификатора мы будем выбирать один элемент из этого множества — *корень* множества. Гарантируется, что для одного и того же множества представитель будет возвращаться один и тот же.

Для нахождения представителя достаточно подняться вверх по родительским ссылкам до тех пор, пока не наткнемся на корень.

Мы будем просто пытаться не допускать чрезмерно длинных веток в дереве (*сжатие путей*). После того, как представитель таки будет найден, мы для каждой вершины по пути от X к корню изменим предка на этого самого представителя. То есть фактически переподвесим все эти вершины вместо длинной ветви непосредственно к корню. Таким образом, реализация операции Find становится двухпроходной.



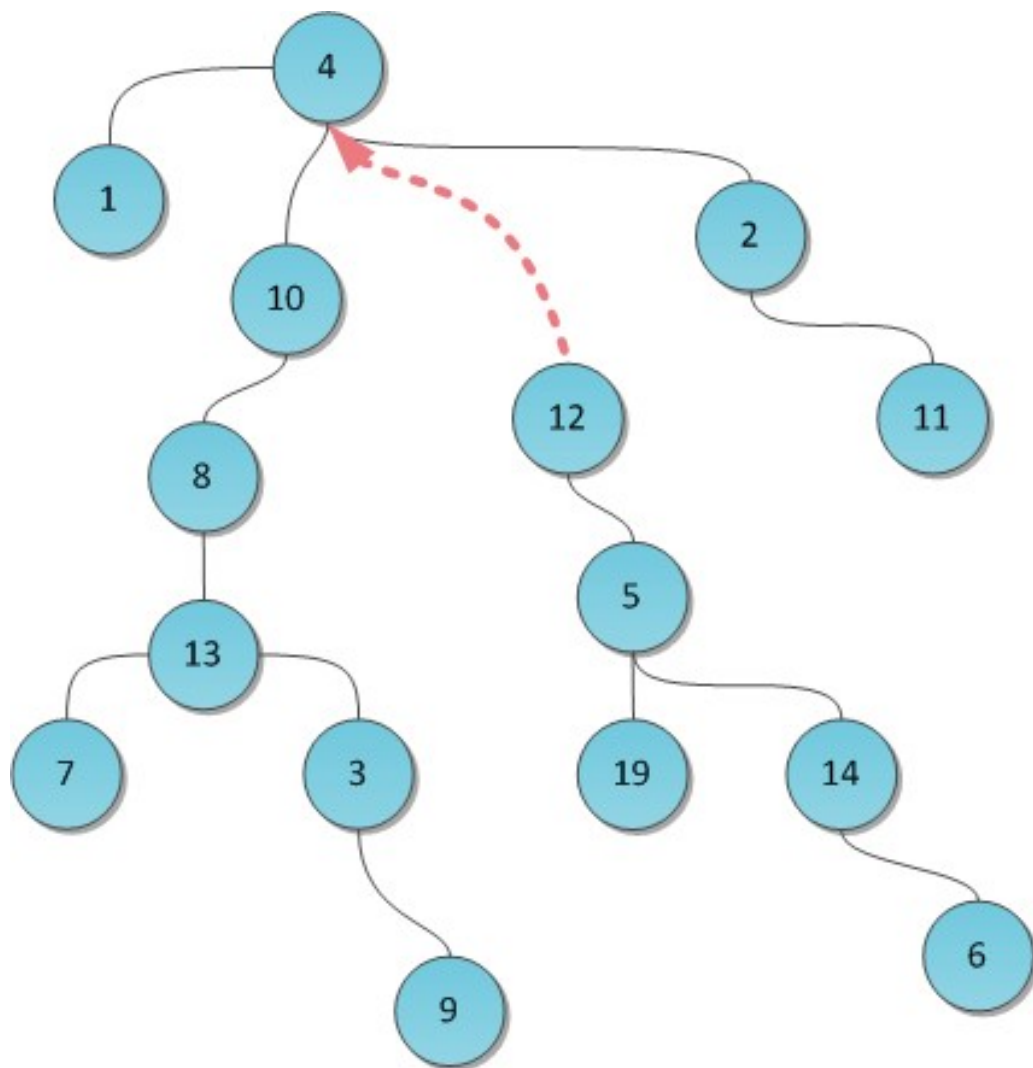
UnionBranch

Найдем для начала корни обоих сливаемых деревьев с помощью уже написанной функции `GetRoot`. Реализация хранит только ссылки на непосредственных родителей, для слияния деревьев достаточно было бы просто подвесить один из корней (а с ним и все дерево) сыном к другому. Таким образом все элементы этого дерева автоматически станут принадлежать другому — и процедура поиска представителя будет возвращать корень нового дерева.

Будем хранить помимо предков еще один массив **Rank**. В нем для каждого дерева будет храниться верхняя граница его высоты — то есть длиннейшей ветви в нем. Для каждого корня в массиве `Rank` будет записано число, гарантированно больше или равное высоте его дерева.

Теперь легко принять решение о слиянии: чтобы не допустить слишком длинных ветвей, будем подвешивать более низкое дерево к более высокому. Если их высоты равны — не играет роли, кого подвешивать к кому. Но в последнем случае новоиспеченному корню надо не забыть увеличить Rank.

Пример, с параметрами 8 и 19:



Тестирование

Содержимое файла:	Вывод:
ivan peter john	Список всех жителей:
anna peter	0 ivan
peter jack	1 peter
igor gorge	2 john
gorge olga	3 anna
dasha andrew maria	4 jack
nikolay andrew maria	5 igor
gennadiy dascha	6 gorge
alex nikolay	7 olga
	8 dascha
	9 andrew
	10 maria
	11 nikolay
	12 gennadiy
	13 alex
	Таблица отношений в индексах жителей:
	0 1
	0 2

	3 1
	1 4
	5 6
	6 7
	8 9
	8 10
	11 9
	11 10
	12 8
	13 11
	Результат работы алгоритма поиска непересекающихся множеств:
	0 ivan
	0 peter
	0 john
	0 anna
	0 jack
	5 igor
	5 gorge
	5 olga
	8 dasha
	8 andrew
	8 maria
	8 nikolay

8 gennadiy

8 alex

Все пары родственников:

ivan - peter

ivan - john

ivan - anna

ivan - jack

peter - john

peter - anna

peter - jack

john - anna

john - jack

anna - jack

igor - gorge

igor - olga

gorge - olga

dasha - andrew

dasha - maria

dasha - nikolay

dasha - gennadiy

dasha - alex

andrew - maria

andrew - nikolay

	<p>andrew - gennadiy</p> <p>andrew - alex</p> <p>maria - nikolay</p> <p>maria - gennadiy</p> <p>maria - alex</p> <p>nikolay - gennadiy</p> <p>nikolay - alex</p> <p>gennadiy - alex</p>
<p>Содержимое файла:</p> <p>ivan peter john</p> <p>anna peter</p> <p>peter jack</p> <p>igor gorge</p> <p>lena vika</p>	<p>Вывод:</p> <p>Список всех жителей:</p> <p>0 ivan</p> <p>1 peter</p> <p>2 john</p> <p>3 anna</p> <p>4 jack</p> <p>5 igor</p> <p>6 gorge</p> <p>7 lena</p> <p>8 vika</p> <p>Таблица отношений в индексах</p>

жителей:

0 1

0 2

3 1

1 4

5 6

7 8

Результат работы алгоритма
поиска непересекающихся
множеств:

0 ivan

0 peter

0 john

0 anna

0 jack

5 igor

5 gorge

7 lena

7 vika

Все пары родственников:

ivan - peter

ivan - john

ivan - anna

	ivan - jack peter - john peter - anna peter - jack john - anna john - jack anna - jack igor - gorge lena – vika
Содержимое файла: ivan	Вывод: Список всех жителей: 0 ivan Таблица отношений в индексах жителей: Результат работы алгоритма поиска непересекающихся множеств: 0 ivan Все пары родственников:

Вывод

В данной работе было создана программа, которая по данным родственным связям находит всех родственников, используя систему непересекающихся множеств.

Исходный код программы

```
#include <fstream>
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <vector>

int GetRoot(int x, std::vector<int>* sets) //
для нахождения представителя, подняться вверх
по родительским ссылкам до тех пор, пока не
наткнемся на корень
//для каждой вершины по пути от X к корню
изменим предка на этого самого представителя
{
    return ((*sets)[x] == x) ? x : (*sets)[x]
= GetRoot((*sets)[x], sets);
}

void UnionBranch(int x, int y,
std::vector<int>* sets, std::vector<int>*
ranks)
{
```

```

        if ( (x = GetRoot(x, sets)) == (y =
GetRoot(y, sets)) )
            return;

        if ( (*ranks)[x] < (*ranks)[y] )
//подвешиваем более низкое дерево к более
высокому
            (*sets)[x] = y;
        else
        {
            (*sets)[y] = x;
            if ( (*ranks)[x] == (*ranks)[y] )
                (*ranks)[x]++;
        }
    }
}

```

```

int GetManID(std::string name,
std::vector<std::string>* people, int
peopleCount)
{
    for (int i = 0; i < peopleCount; i++)
        if ((*people)[i] == name)
            return i;
    //в цикле по people сравниваем name с
people[i]. если равно, то возвращаем i
}

```

```

void
ReadAndWritePeople(std::vector<std::string>*
people, std::vector<int>* parents,
std::vector<int>* children, int* peopleCount,
int* relationsCount)

```

```

{
    // в цикле считываем строки
    std::string str;
    std::ifstream file("relations.txt",
std::ios::in);
    while(!file.eof())
    {
        std::getline(file, str);
        char* cstr = new char[str.length()+1];
        strcpy(cstr, str.c_str());
        char* name = new char[20];
        name = strtok (cstr, " ");

        int flagChild = 0;
        std::string parentName;
        int parentID;

        // ищем имена в people[]
        // если имени нет - добавить

        while (name != NULL)
        {
            int flag = 1;
            for(int i = 0; i < *peopleCount;
i++){
                if(strcmp(name, (*people)
[i].c_str()) == 0) { flag = 0;    break; }
            }
            if(flag){
                people->push_back(name);
                //people[*peopleCount] =
name;

                (*peopleCount)++;
            }
        }
    }
}

```

```

    }

    if(flagChild){          // если
считали второе и последующие имена в строке,
то добавляем отношение
        parents->push_back(parentID);
//(индекс первого имени в строке - индекс
считанного имени)
        children-
>push_back(GetManID(name, people,
*peopleCount));
        (*relationsCount)++;
    }
    else
    {
        parentName = name;
        parentID =
GetManID(parentName, people,
*peopleCount); //функция поиска индекса имени
в массиве имен
    }

    name = new char[20];
    name = strtok (NULL, " ");
    flagChild = 1;
}
}

std::cout << "\nСписок всех жителей:" <<
std::endl;
for (int i = 0; i < *peopleCount; i++){
    std::cout << i << " " << (*people)[i]
<< std::endl;

```

```

    }

    std::cout << "\nТаблица отношений в
индексах жителей:" << std::endl;
    for (int i = 0; i < *relationsCount; i++){
        std::cout << (*parents)[i] << " " <<
(*children)[i] << std::endl;
    }

}

void
PrintAllRelatives(std::vector<std::string>*
people, std::vector<int>* peopleSets, int
peopleCount)
{
    std::cout << "\nРезультат работы алгоритма
поиска непересекающихся множеств:" <<
std::endl;

    for (int i = 0; i < peopleCount; i++)
        std::cout << (*peopleSets)[i] << " " +
(*people)[i] << std::endl;

    std::cout << "\nВсе пары родственников:"
<< std::endl;

    for (int i = 0; i < peopleCount-1; i++)
    {
        for (int j = i+1; j < peopleCount; j+
+)
        {

```



```

        if ((*peopleSets)[i] ==
(*peopleSets)[j])
            std::cout << (*people)[i] + "
- " + (*people)[j] << std::endl;
        }
    }
}

```

```

int main()
{
    //string* people = new string[SIZE];
    //список всех имен

    std::vector<std::string> people;

    std::vector<int> peopleSets; //массив,
хранящий для каждой вершины дерева её
непосредственного предка
    std::vector<int> peopleSetsRanks; // для
каждого дерева будет храниться верхняя граница
его высоты

    std::vector<int> parents; //индексы имен
из people в двух массивах показывают родство
    std::vector<int> children;
    int peopleCount = 0; //количество жителей
    int relationsCount = 0; //количество
связей

    ReadAndWritePeople(&people, &parents,
&children, &peopleCount, &relationsCount); //
считывание имен и заполнение связей родства

```

```

        for (int i = 0; i < peopleCount; i++)
//для каждого элемента X, создать множество
размера 1 из самого себя.
        {
            peopleSets.push_back(i);
            peopleSetsRanks.push_back(0);
        }

        for (int i = 0; i < relationsCount; i++)
//объединить два множества, в которых
лежат элементы X и Y, в одно новое
        {
            UnionBranch(parents[i], children[i],
&peopleSets, &peopleSetsRanks);
        }
        for (int i = 0; i < peopleCount; i++)
//возвратить идентификатор множества, которому
принадлежит элемент X (обновляем)
        {
            peopleSets[i] = GetRoot(i,
&peopleSets);
        }

        PrintAllRelatives(&people, &peopleSets,
peopleCount); //печать пар родственников

        return 0;
    }

    for (int i = 0; i < peopleCount; i++)
        cout << peopleSets[i] << " " +
people[i] << endl;

```

```

        cout << "\nВсе пары родственников:\n" <<
endl;

        for (int i = 0; i < peopleCount-1; i++)
        {
            for (int j = i+1; j < peopleCount; j+
+)
            {
                if (peopleSets[i] ==
peopleSets[j])
                    cout << people[i] + " - " +
people[j] << endl;
            }
        }

    }

int main()
{
    string[] people = new string[SIZE];
    //список всех имен

    int* peopleSets = new int[SIZE];
    //массив, хранящий для каждой вершины дерева
её непосредственного предка
    int* peopleSetsRanks = new int[SIZE]; //
для каждого дерева будет храниться верхняя
граница его высоты

    int* parents = new int[SIZE]; //индексы
имен из people в двух массивах показывают
родство

```

```
int* children = new int[SIZE];  
int peopleCount = 0; //количество жителей  
int relationsCount = 0; //количество  
связей
```

```
ReadAndWritePeople(people, parents,  
children, &peopleCount, &relationsCount);  
//считывание имен и заполнение связей родства
```

```
for (int i = 0; i < SIZE; i++) //для  
каждого элемента X, создать множество размера  
1 из самого себя.
```

```
{  
    peopleSets[i] = i;  
    peopleSetsRanks[i] = 0;  
}
```

```
for (int i = 0; i < relationsCount; i+  
+) //объединить два множества, в которых  
лежат элементы X и Y, в одно новое
```

```
{  
    UnionBranch(parents[i], children[i],  
peopleSets, peopleSetsRanks);  
}
```

```
for (int i = 0; i < relationsCount; i+  
+) //возвратить идентификатор множества,  
которому принадлежит элемент X
```

```
{  
    peopleSets[i] = GetRoot(i,  
peopleSets);  
}
```

```
        PrintAllRelatives(people, peopleSets,  
peopleCount); //печать пар родственников  
        return 0;  
    }
```